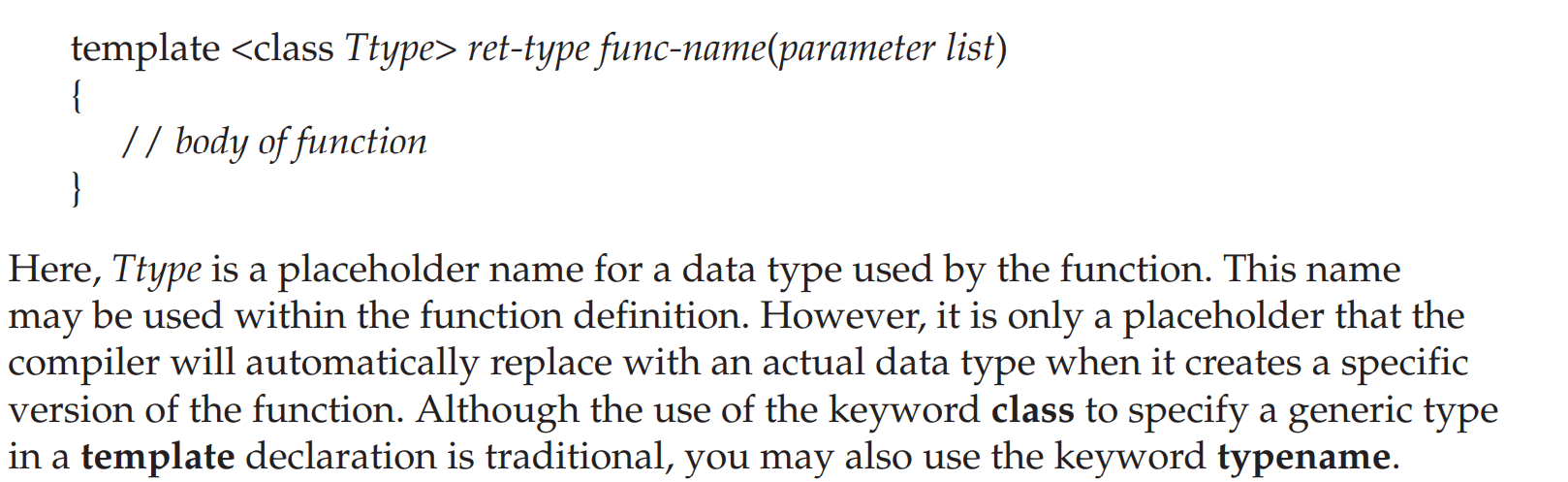
Generic Functions

A generic function defines a general set of operations that will be applied to various types of data.

The type of data that the function will operate upon is passed to it as a parameter

Through a generic function, a single general procedure can be applied to a wide range of data.

A generic function is created using the keyword template. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a template function definition is shown here:



// Function template example.

#include <iostream>

using namespace std;

// This is a function template.

template <class X> void swapargs(X &a, X &b)

{

X temp;

temp = a;

a = b;

b = temp;

}

int main()

{

int i=10, j=20;

double x=10.1, y=23.3;

char a='x', b='z';

cout << "Original i, j: " << i << ' ' << j << '\n';

cout << "Original x, y: " << x << ' ' << y << '\n';

cout << "Original a, b: " << a << ' ' << b << '\n';

swapargs(i, j); // swap integers

swapargs(x, y); // swap floats

swapargs(a, b); // swap chars

cout << "Swapped i, j: " << i << ' ' << j << '\n';

cout << "Swapped x, y: " << x << ' ' << y << '\n';

cout << "Swapped a, b: " << a << ' ' << b << '\n';

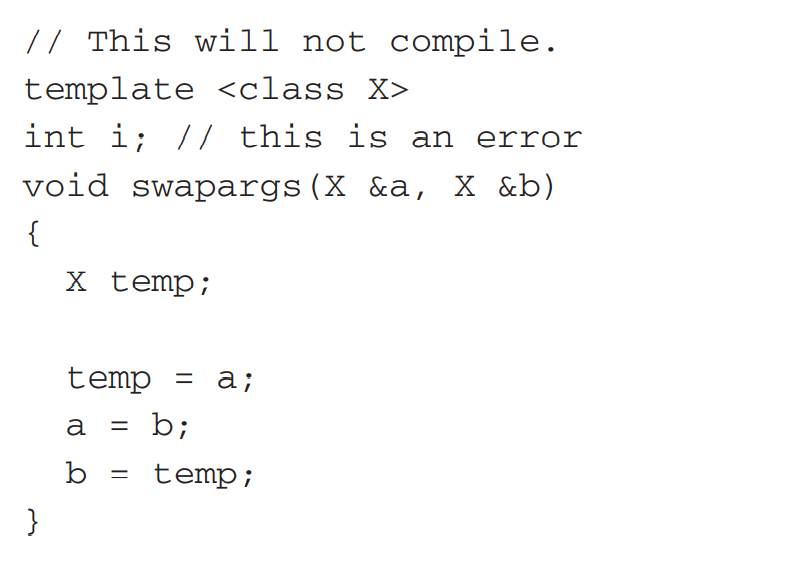
return 0;

}

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, X is a generic type that is used as a placeholder. After the template portion, the function swapargs( ) is declared, using X as the data type of the values that will be swapped.

In main( ), the swapargs( ) function is called using three different types of data: ints, doubles, and chars. Because swapargs( ) is a generic function, the compiler automatically creates three versions of swapargs( ): one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

Wrong declaration:



A Function with Two Generic Types

//A Function with Two Generic Types//

#include <iostream>

using namespace std;

template <class type1, class type2>

void myfunc(type1 x, type2 y)

{

cout << x << ' ' << y << '\n';

}

int main()

{

myfunc(10, "I like C++");

myfunc(98.6, 19L);

return 0;

}

//Explicitly Overloading a Generic Function//

#include <iostream>

using namespace std;

template <class X> void swapargs(X &a, X &b)

{

X temp;

temp = a;

a = b;

b = temp;

cout << "Inside template swapargs.\n";

}

// This overrides the generic version of swapargs() for ints.

void swapargs(int &a, int &b)

{

int temp;

temp = a;

a = b;

b = temp;

cout << "Inside swapargs int specialization.\n";

}

int main()

{

int i=10, j=20;

double x=10.1, y=23.3;

char a='x', b='z';

cout << "Original i, j: " << i << ' ' << j << '\n';

cout << "Original x, y: " << x << ' ' << y << '\n';

cout << "Original a, b: " << a << ' ' << b << '\n';

swapargs(i, j); // calls explicitly overloaded swapargs()

swapargs(x, y); // calls generic swapargs()

swapargs(a, b); // calls generic swapargs()

cout << "Swapped i, j: " << i << ' ' << j << '\n';

cout << "Swapped x, y: " << x << ' ' << y << '\n';

cout << "Swapped a, b: " << a << ' ' << b << '\n';

return 0;

}

A Generic Sort

Sorting is exactly the type of operation for which generic functions were designed. Within wide latitude, a sorting algorithm is the same no matter what type of data is being sorted. The following program illustrates this by creating a generic bubble sort. While the bubble sort is a rather poor sorting algorithm, its operation is clear and uncluttered and it makes an easy-to-understand example. The bubble( ) function will sort any type of array. It is called with a pointer to the first element in the array and the number of elements in the array.

// A Generic bubble sort.

#include <iostream>

using namespace std;

template <class X> void bubble(

X \*items, // pointer to array to be sorted

int count) // number of items in array

{

register int a, b;

X t;

for(a=1; a<count; a++)

for(b=count-1; b>=a; b--)

if(items[b-1] > items[b]) {

// exchange elements

t = items[b-1];

items[b-1] = items[b];

items[b] = t;

}

}

int main()

{

int iarray[7] = {7, 5, 4, 3, 9, 8, 6};

double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

int i;

cout << "Here is unsorted integer array: ";

for(i=0; i<7; i++)cout << iarray[i] << ' ';

cout << endl;

cout << "Here is unsorted double array: ";

for(i=0; i<5; i++)

cout << darray[i] << ' ';

cout << endl;

bubble(iarray, 7);

bubble(darray, 5);

cout << "Here is sorted integer array: ";

for(i=0; i<7; i++)

cout << iarray[i] << ' ';

cout << endl;

cout << "Here is sorted double array: ";

for(i=0; i<5; i++)

cout << darray[i] << ' ';

cout << endl;

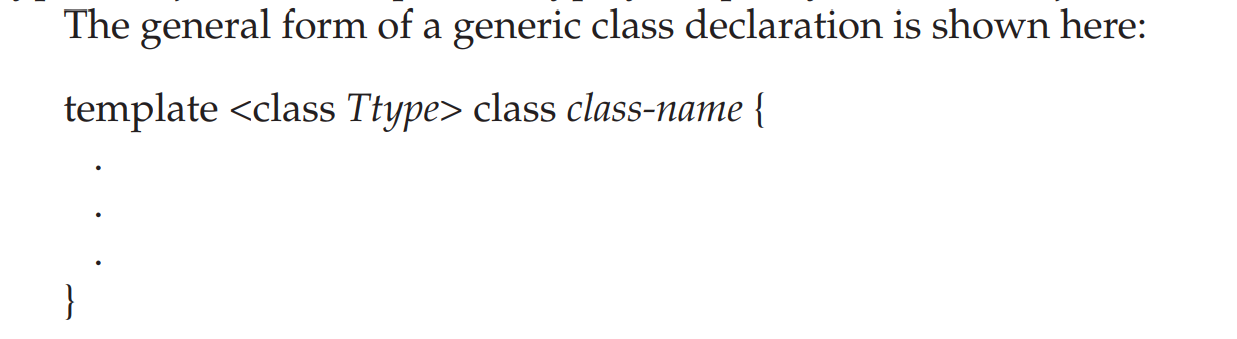
return 0;

}

Generic Classes

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.



// This function demonstrates a generic stack.

#include <iostream>

using namespace std;

const int SIZE = 10;

// Create a generic stack class

template <class StackType> class stack {

StackType stck[SIZE]; // holds the stack

int tos; // index of top-of-stack

public:

stack() { tos = 0; } // initialize stack

void push(StackType ob); // push object on stack

StackType pop(); // pop object from stack

};

// Push an object.

template <class StackType> void stack<StackType>::push(StackType ob)

{

if(tos==SIZE) {

cout << "Stack is full.\n";

return;

}

stck[tos] = ob;

tos++;

}

// Pop an object.

template <class StackType> StackType stack<StackType>::pop()

{

if(tos==0) {

cout << "Stack is empty.\n";

return 0; // return null on empty stack

}

tos--;

return stck[tos];

}

int main()

{

// Demonstrate character stacks.

stack<char> s1, s2; // create two character stacks

int i;

s1.push('a');

s2.push('x');

s1.push('b');

s2.push('y');

s1.push('c');

s2.push('z');

for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";

for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

// demonstrate double stacks

stack<double> ds1, ds2; // create two double stacks

ds1.push(1.1);

ds2.push(2.2);

ds1.push(3.3);

ds2.push(4.4);

ds1.push(5.5);

ds2.push(6.6);

for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";

for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

return 0;

}

//Two Generic Data Types of Calss//

/\* This example uses two generic data types in a

class definition.

\*/

#include <iostream>

using namespace std;

template <class Type1, class Type2> class myclass

{

Type1 i;

Type2 j;

public:

myclass(Type1 a, Type2 b) { i = a; j = b; }

void show() { cout << i << ' ' << j << '\n'; }

};

int main()

{

myclass<int, double> ob1(10, 0.23);

myclass<char, char \*> ob2('X', "Templates add power.");

ob1.show(); // show int, double

ob2.show(); // show char, char \*

return 0;

}

**Applying Template Classes:**

A Generic Array Class To illustrate the practical benefits of template classes, let's look at one way in which they are commonly applied. As you saw in Chapter 15, you can overload the [ ] operator. Doing so allows you to create your own array implementations, including "safe arrays" that provide run-time boundary checking. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded [ ] subscripting operator, then you can intercept an out-of-range index.

// A generic safe array example.

#include <iostream>

#include <cstdlib>

using namespace std;

const int SIZE = 10;

template <class AType> class atype {

AType a[SIZE];

public:

atype() {

register int i;

for(i=0; i<SIZE; i++) a[i] = i;

}

AType &operator[](int i);

};

// Provide range checking for atype.

template <class AType> AType &atype<AType>::operator[](int i)

{

if(i<0 || i> SIZE-1) {

cout << "\nIndex value of ";

cout << i << " is out-of-bounds.\n";

exit(1);

}

return a[i];

}

int main()

{

atype<int> intob; // integer array

atype<double> doubleob; // double array

int i;

cout << "Integer array: ";

for(i=0; i<SIZE; i++) intob[i] = i;

for(i=0; i<SIZE; i++) cout << intob[i] << " ";

cout << '\n';

cout << "Double array: ";

for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;

for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";

cout << '\n';

intob[12] = 100; // generates runtime error

return 0;

}

Using Non-Type Arguments with Generic Classes

In the template specification for a generic class, you may also specify non-type arguments. That is, in a template specification you can specify what you would normally think of as a standard argument, such as an integer or a pointer. The syntax to accomplish this is essentially the same as for normal function parameters: simply include the type and name of the argument. For example, here is a better way to implement the safe-array class presented in the preceding section. It allows you to specify the size of the array.

// A generic safe array example.

#include <iostream>

#include <cstdlib>

using namespace std;

const int SIZE = 10;

template <class AType> class atype {

AType a[SIZE];

public:

atype() {

register int i;

for(i=0; i<SIZE; i++) a[i] = i;

}

AType &operator[](int i);

};

// Provide range checking for atype.

template <class AType> AType &atype<AType>::operator[](int i)

{

if(i<0 || i> SIZE-1) {

cout << "\nIndex value of ";

cout << i << " is out-of-bounds.\n";

exit(1);

}

return a[i];

}

int main()

{

atype<int> intob; // integer array

atype<double> doubleob; // double array

int i;

cout << "Integer array: ";

for(i=0; i<SIZE; i++) intob[i] = i;

for(i=0; i<SIZE; i++) cout << intob[i] << " ";

cout << '\n';

cout << "Double array: ";

for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;

for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";

cout << '\n';

intob[12] = 100; // generates runtime error

return 0;

}